Heterogeneous System for LLM Inference Acceleration

Yuhao LIU
School of Data Science
Chinese University of Hong Kong, Shenzhen
224040365@link.cuhk.edu.cn

***Abstract*—Nowadays, Large Language Models (LLMs) are changing the world with their powerful capabilities on various tasks In order to improve their effectiveness, they are becoming larger and larger with massive parameters. On the other hand, there are more and more diverse scenarios where there is no available high-end computation resources(GPUs). Therefore, its inference becomes an expensive problem. Traditionally, GPU is assumed as the only appropriate device since most of computation is the tensor multiplication GPU excels in, while CPU's computing power is negligible. But for the inference, the performance bottleneck of GPU is the slow data loading via PCIe bus. The introduction of CPU computation with data held in CPU memory would shorten the overall execution time if using a heterogeneous system where CPU and GPU are running in parallel, which is the potential to accelerate inference under the assistance of CPU.**

This survey presents several existing methods which come up with different scheduling strategies to utilize CPU-GPU systems, and evaluate their performance. Finally, it offers several possible directions for future exploration.

***Index Terms*—Heterogeneous system, LLM inference**

I. Introduction

Large Language Models[1] (LLMs) are becoming more and more popular among the world due to their versatile capabilities and outstanding performance for diverse tasks. OpenAI's GPT [2], Google's PaLM [3], Meta's Llama [4] are famous examples which are used at every field almost everyday. Usually, the large models are trained at large and GPU clusters with powerful computation for many days before being deployed for miscellaneous inference applications [5, 6]. One of the important characters of LLMs' inference is response time. The reason for it is that, one model could be used in an interactive scenario where it is serving a few users. Therefore, the response latency — output tokens per second for each user — is more important than the system's throughput. Recently, the large models have been having larger and larger parameters (millions, billions, or even trillions), which makes them more efficient for inference.

Based on the above description, while the LLMs are becoming more and more powerful, its inference is a challenging and expensive problem because of the high computation and extreme memory requirements and economic cost. Even the amount of computation during inference is much less than that of training, which allows us to deploy it in lower-end GPUs, the demand for parameters and working set makes the GPU memory insufficient to fit the model. For instance, the GPT-175B requires 325GB of GPU memory to load model weights. To fit this model into GPUs, we need at least five A100 (80GB) GPUs and complicated computation strategies [7]. If there are variable length KV cache (key-value cache) tensors, the memory shortage issue becomes even worse. Another issue is, serving a GPT-3 model for a production-scale application could cost between $100,000 to $300,000 monthly.

On the other hand, it is not uncommon to find that the large model inference is not only run on resource-rich devices, but also on diverse system setting where there is no high-end GPUs, like edge computing system and mobile devices. Therefore, under these environments the memory constraint is much severe. Thus, lowering LLM inference resource requirements is a urgent problem and has caught a lot of attention.

Meanwhile, the CPU memory is much larger than GPU's. The figure for that can reach to hundreds of Gigabytes. So placing a large number of inference's data into CPU memory is a common way. And transferring data among CPU and GPU memory happens frequently. However, the data loading and offloading relies on PCIe I/O bus. Compared with GPU's memory, the latter one's bandwidth is much lower, with 16GB/s to 600GB/s. The data stall could arise from the low PCIe bandwidth. The data used at different stages would take a long time to load via PCIe bus, but GPU can consume it all within a much shorter time period. This progress would not stop until the whole inference is completed. As a result, GPU is running at a fraction of its full speed due to the PCIe's low bandwidth. Thus this bottleneck prevents the deployment of LLMs on diverse environments, affecting users' experience.

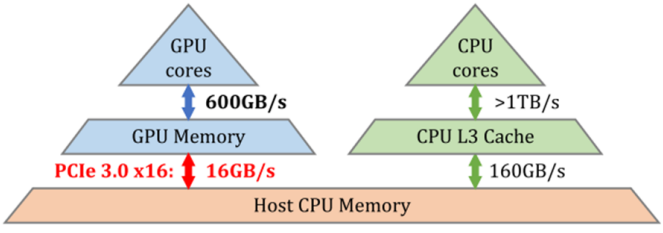


Fig. 1. The GPU-CPU Architecture.

Surprisingly, the idea of utilizing CPU and GPU simultaneously to perform inference and assist each other is emerging. Although CPU does not have the strong computation power as GPU, facing the memory-constrained GPU waiting for new data, CPU can share portion of computation tasks since

the bandwidth of L3 cache is much higher than PCIe bus (approximately 160Gb/s, as shown in Fig. 1). It is possible to assign computation onto both GPU and CPU, thus it can alleviate the PCIe bus overhead. This survey would review some exiting models and methods to accelerate LLMs' inference, and present some novel methods utilizing CPU-GPU system and evaluate corresponding performance. In the end, there would be some considerations for future research.

II. Related work

Recent advancements in LLMs inference are showing the growing importance of optimizing both system and algorithm aspects of LLM workloads.

Several specialized systems for LLM inference have emerged in recent years, such as FasterTransformer[8], Orca[9], LightSeq[10], TurboTransformers[11], DeepSpeed Inference[12], and Hugging Face Accelerate[13], among others. These systems are mainly focused on latency-oriented scenarios with high-end accelerators, which limits their use in throughput-oriented inference on more easily accessible, commodity hardware.

One of the critical techniques to enable LLM inference on commodity hardware is offloading, which involves shifting parts of the computation to other devices like CPUs or distributed systems. However, among the existing systems, only DeepSpeed Zero-Inference and Hugging Face Accelerate support offloading. Many of the current systems rely on offloading techniques adapted from training systems (e.g., DeepSpeed Zero etc.), but they overlook the unique computational properties of generative inference, particularly in how to optimize the scheduling of I/O traffic. The Petals[14] framework takes a different approach by exploring collaborative computing, which aims to enable LLM inference on more accessible hardware.

In terms of algorithmic optimizations, sparsification and quantization have been widely adopted to accelerate LLM inference and reduce memory usage. Works on sparsification[15] and quantization[16] show that weights and activations can be compressed effectively, often down to 3-8 bits for weights and activations. In FlexGen[17], the authors propose a novel approach by compressing both weights and the KV cache to 4 bits, which they combine with offloading techniques to further optimize inference.

There is also several work on memory optimizations and offloading in the broader context of training [18] and linear algebra optimizations [19]. These works provide a foundation for optimizing memory usage and computation during the inference process, contributing to the efficiency of LLM systems. And vLLM [20] present PagedAttention mechanism to addresses memory allocation challenges in serving LLMs by achieving near-zero waste in KV cache memory. It uses block-level memory management and preemptive request scheduling, both co-designed with PagedAttention, enabling more efficient use of memory resources.

III. Preliminaries

This section would describe the concrete LLM inference problem and basic background.

A. LLM Structure

The typical LLM structure is shown in Fig. 2 [21], where the model has 4 layers and generate 3 tokens per prompt.

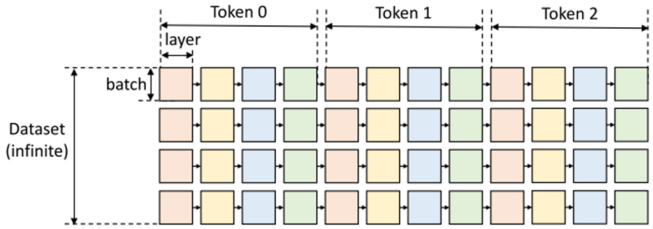


Fig. 2. Computational graph of LLM inference

In the LLM structure, each square represents the computation of a prompt for a layer. For the computation of the different prompts, the squares in the same column would use the same layer weights. During the inference, the model would produce answer for each prompt token by token. When a token's computation is completed, this output token would be fed to the first layer so that produce the next token. This process would be repeated until the model generates a terminal token, which shows all token computation is over, or the token length reaches the maximum limitation.

Here are computation rules among the whole process to obey:

- Sequential computation: A square can be computed only if all square units to its left have been completed
- All inputs to be loaded: To compute a square, all data needed — weights, activations, cache — must be loaded to the same device.
- KV cache lasting time: A square would generate two outputs after its computation: activations and KV cache. The activations should be stored until its right square unit is computed. The KV cache must be kept until we complete the last square's computation on the same row.
- Memory capacity: During inference, the size of active tensor would be limited to the device memory.

B. Problem Formulation

A traditional system has a three-level memory hierarchy formed by GPU, CPU, and disk. The GPU and CPU are capable of performing computations, and the disk is used to store large number of data. GPU has the smallest but fastest memory, while the disk has the largest but slowest memory. When a LLM cannot fully fit into the GPU's memory, it must be offloaded to secondary storage, and computations are performed in segments by partially loading the LLM.

We can model the generative inference with offloading as a graph traversal problem. For example, Fig. 3 shows two kinds of path to perform the computation. Fig. 3 (a) schedule all computation row by row, while this way cannot reuse the same

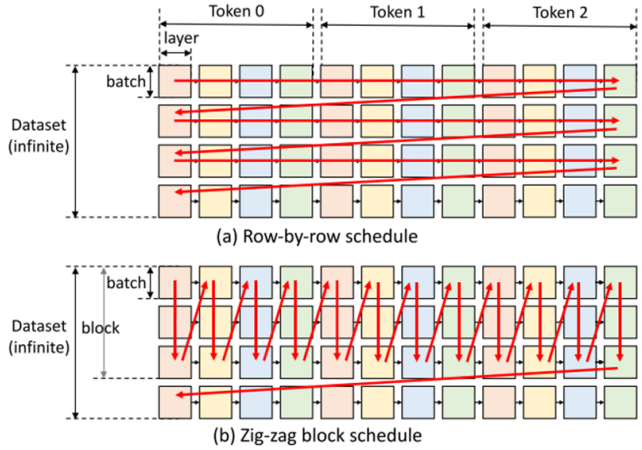


Fig. 3. Two computation schemes. The red arrows denote the computation order.

data in the same layer. Therefore it would cost a lot of time to re-load weights. Fig. 3 (b) would perform all computation for the same layer before it moves to the next layer's computation. As a result it can reuse the same data needed for computation and save loading time, which is called Zig-zag block schedule. Thus, our goal is to find a valid path that minimizes the total execution time, which includes the computation and cost of moving tensor among different devices via I/O.

C. Inference Stages

The generation process consists of two phases: prefilling and decoding, as illustrated in Fig. 4. In the prefilling phase, the user's input prompt is provided to the model, and the model would initialize the KV cache for each transformer layer. The KV cache plays a crucial role in the subsequent decoding phase. During decoding, the model utilizes the KV cache to generate output tokens in a sequential manner. Once a token is produced, it is reintroduced into the model to update the KV cache. The updated cache is then used to generate the following token.

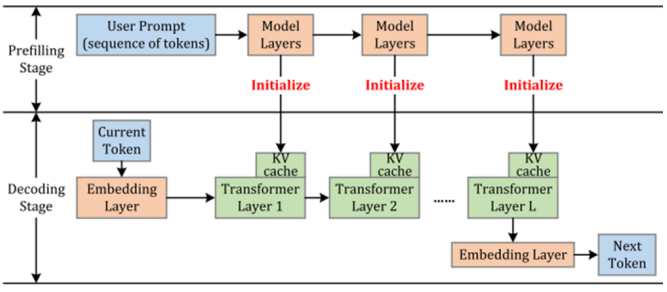


Fig. 4. The LLM inference process

Fig. 4 provides additional insight into the decoding stage of a typical LLM inference process. The architecture of such models is generally composed of multiple stages, starting with an input embedding layer, followed by several transformer layers, and concluding with an output embedding layer.

The decoding process begins with the input embedding layer, which converts the raw input tokens into dense vectors. This transformation is achieved by row indexing in a weight matrix that captures the semantic representation of the input features. The embedding layer helps to map the discrete tokens into continuous representations, which are more suitable for processing by the subsequent layers. These embeddings, containing contextual information, serve as the foundation for the model's understanding of the input sequence.

Following the embedding layer, the model enters the core of its architecture: the transformer layers. Each transformer layer consists of several subcomponents, including multi-head attention and feedforward networks. These subcomponents enable the model to capture intricate dependencies within the input sequence, facilitating the generation of coherent and contextually relevant output tokens. At each transformer layer, multiple matrix multiplications are performed, which contribute significantly to the computational cost and latency during decoding. As the number of transformer layers increases, the time complexity of these operations grows, making them the primary bottleneck in the overall decoding stage.

After passing through the transformer layers, the output is processed by the final output embedding layer, which maps the hidden representations back into the vocabulary space. The generated tokens are then used as the model's output. Crucially, the model also maintains a KV cache during the decoding stage. This cache stores the intermediate results from previous steps, which helps the model efficiently generate the next token by using these cached values rather than recalculating them. This caching mechanism plays a crucial role in optimizing the inference process, especially when dealing with large-scale models.

Transformer layer is a time-consuming stage due to the massive matrix multiplication, and it dominates the latency of decoding stage. Therefore it is necessary to go deep into it. Fig. 5 shows the basic data flow in a transformer layer. There are seven operations in each layer: Q, K, V, SCORE, OUT, MLP1, and MLP2. Table I shows more details on description and data dependency.

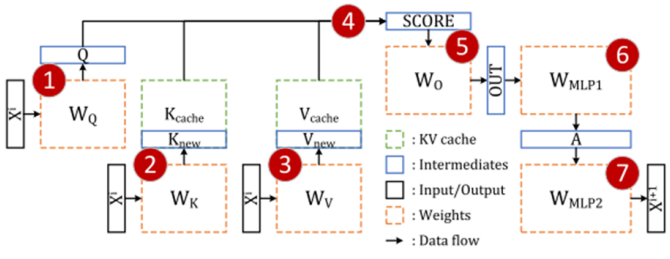


Fig. 5. Transformer layer

D. Data Transfer Trick

Because it is unavoidable to move data among devices, the mechanism of transferring is crucial. Here would review tow common ways, and Fig. 6 shows them:

TABLE I
OPERATIONS IN TRANSFORMER LAYER.

Operation	Description	Dependency
Q	$q = X^i \times W_Q^i$	-
K	$k = X^i \times W_K^i$	-
V	$v = X^i \times W_V^i$	-
SCORE	score = Attention(q, APPEND(K,k), APPEND(V,v))	Q,K,V
OUT	$O = score \times W_O^i$	SCORE
MLP1	$A = O \times W_{MLP1}^i$	OUT
MLP2	$X^{i+1} = A \times W_{MLP2}^i$	MLP1

- The GPU Direct Memory Access (DMA) engine is a typically efficient way to transfers large amounts of data between CPU memory and GPU memory. And it is non-preemptive, meaning that subsequent DMA transfers are blocked until all earlier-issued transfers are completed. This non-preemptive behavior can lead to unpredictable delays in the transfer of intermediate results. Some background transfers can block the required DMA channels, thus introducing latency in the execution pipeline.
- Zero-Copy mechanism. This mechanism enables direct read/write operations between the GPU cores and page-locked CPU memory over the PCIe bus, bypassing the need for intermediate memory copies. Limited by PCIe bandwidth, this approach is suitable to moving smaller scale of data. By using Zero-Copy, the intermediate result transfers would not be blocked by ongoing background DMA operations.

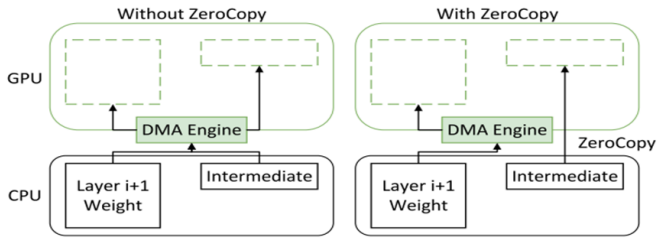


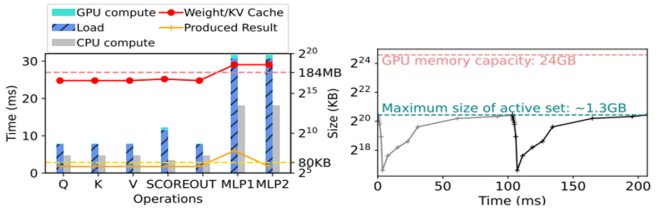
Fig. 6. GPU DMA and Zero-Copy

E. Potential of Leveraging CPU

Although the data loading does cost significantly much time than the GPU computing time, we can find a substantial reduction in data size after completing each operation. For instance, we can analyze the space and time breakdown of each operation as shown in Fig. 7 (a). An obvious observation is the size of each operation's input and output tensors is much smaller than that of weights and KV cache involved in the computation. If some operations can be offloaded to the host CPU, we can save the significant amount of time by transferring only a small amount of intermediate results to CPU instead of large data (weights and KV cache) to the GPU, which can reduce the overall execution time.

F. Judiciously Using GPU Memory

Memory shortage should not be overlooked, which limits the smooth GPU computation. On the other hand, the size of current layer's active tensor and the space allocated for the next layer (these two pieces of space is called working set) is much smaller than the whole GPU memory. Fig. 7 (b) shows the periodic change of active tensor in a NVIDIA A10G card with 24GB memory as the different operation computing. We can find a sudden drop down of the active tensor size. It is because the computation of current layer has been completed, and its weights and KV cache can be freed. This fact shows a considerable amount of memory space we can utilize.



(a) Time breakdown and data size (b) Active tensor size

Fig. 7. Transformer layer breakdown(prompt length: 896, decode length: 128)

IV. PROPOSED METHODS

The efficient computation among all layers concerns two parts:

- Scheduling. Usually a set of rules or policies about the decisions on computation execution or corresponding data transferring among different devices consists of scheduling. It involves when and which task (computation and data moving) should be assigned to which device. The design of the scheduling may combine math models and optimization algorithm to implement a scheduler to manage task scheduling. Moreover, during the progress of inference, we can perform multiple tasks in parallel. For instance, CPU could make Q operation and GPU can make K operation, while there is background data transferring onto GPU or CPU. Therefore, the overlapping is an critical part of scheduling.
- Tensor placement. It concerns all data storage management. The tensor involved includes weights, KV cache and activations (intermediate result). The strategies of placing data has significant impact on the performance, since weights and KV cache are substantial, which not allows massive placement onto GPU memory. However, we also need to utilize high-level memory to accelerate data access. Thus the balance between the loading and offloading is crucial.

In this context, this report would introduce informed description of three methods—FlexGen, Llama.cpp, and twinPilots. Firstly, here is a briefly explanation on their core concepts. Then more details on each part involved would be represented.

FlexGen [17], a method designed to optimize the memory usage and computation efficiency of transformer models, is a framework for running large models without exceeding memory limits.

Llama.cpp [4] is designed to run large models such as LLaMA models in environments with constrained resources like CPU or lower-end GPUs. And Llama.cpp is a lightweight inference framework optimized for efficiency.

TwinPilots [21], which is a parallel inference technique designed to improve memory management and speed for large-scale language models, leverages parallelism for efficient deployment in multi-GPU environments.

A. Scheduler

The FlexGen method is a GPU-centric pipeline approach where the scheduler focuses on optimizing the inference process by using a cost model. This cost model predicts the latency of transformer layer inference, including both computation and data transfer times. FlexGen aims to find the best balance between computation and data transfer by employing linear programming (as formula (1) shown). The primary objectives of FlexGen scheduling include:

$$\begin{aligned} \min_{p} \quad & \frac{T}{bls} \\ \text{s.t.} \quad & \text{gpu peak memory} < \text{gpu mem capacity} \\ & \text{cpu peak memory} < \text{cpu mem capacity} \\ & \text{disk peak memory} < \text{disk mem capacity} \\ & wg + wc + wd = 1 \\ & cg + cc + cd = 1 \\ & hg + hc + hd = 1 \end{aligned} \tag{1}$$

- Latency Prediction: Estimating the time it will take for each layer's inference, considering both the GPU computation and the necessary data transfers.
- Data Placement Optimization: The scheduler tries to determine the best way to allocate weights, activations, and KV cache across CPU and GPU memory. It aims to minimize memory bottlenecks and maximize the effective use of GPU resources.

The Llama.cpp framework uses a static layer partitioning strategy for scheduling. Fig. 8 represents a simple model for Llama.cpp. In this method, the first few contiguous layers of the model are executed on the CPU, while the remaining layers are handled by the GPU. This approach simplifies the scheduling problem by reducing the complexity of dynamic decisions during runtime, but it may not fully optimize resource utilization in all cases.

- Static Partitioning: By predefining which layers should run on the CPU and which should run on the GPU, Llama.cpp avoids the overhead of dynamic scheduling but may not be as efficient in utilizing both resources.
- Memory Management: This method also ensures that the initial layers, which typically require less memory and computation, are processed on the CPU, leaving more demanding layers to be handled by the GPU.

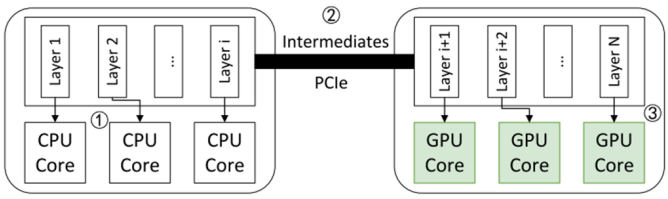


Fig. 8. Data placement of Llama.cpp

The TwinPilots approach introduces a more sophisticated, online scheduler that makes real-time decisions based on the current state of the system. Fig. 9 demonstrates the overview of TwinPilots, whose scheduler consists of two primary components:

- Statistics Tracker: Collects real-time information about the operation execution times on both CPU and GPU. This includes the time it takes to perform computations on the CPU and the time required to load data onto the GPU. By tracking these statistics, the scheduler can dynamically adjust its decisions based on workload characteristics.
- Planner: The planner uses the information collected by the statistics tracker to perform load balancing across the CPU and GPU. It aims to allocate tasks in a way that minimizes the overall execution time by ensuring neither CPU nor GPU is overloaded. The load-balancing problem is modeled as a linear programming (LP) problem, where the objective is to minimize the maximum load on either the CPU or GPU. This is achieved by solving for the optimal allocation of tasks between the two processors.

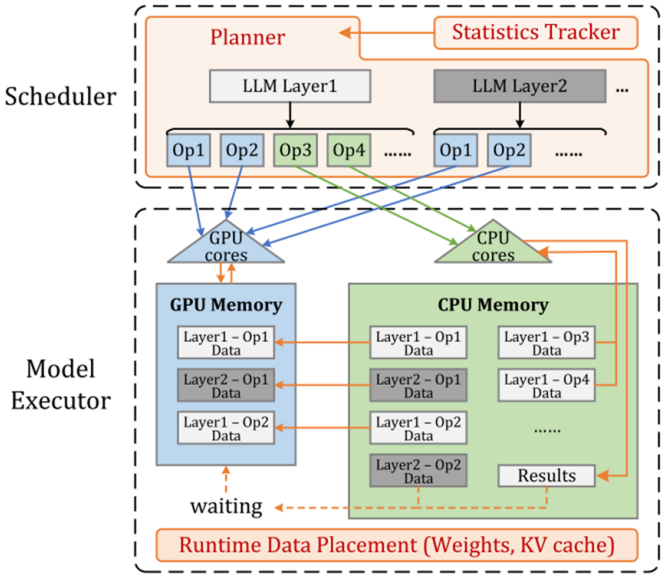


Fig. 9. TwinPilots overview

The load balancing strategy in TwinPilots is based on minimizing the imbalance between the CPU and GPU workloads.

The problem is mathematically formulated as:

- $S_{cpu} = \sum_i N_{cpu_i} \cdot device_i$: The total load on the CPU, where N_{cpu_i} is the number of operations assigned to CPU device i.
- $S_{gpu} = \sum_i N_{gpu_i} \cdot (1 - device_i)$: The total load on the GPU, where N_{gpu_i} represents operations assigned to GPU device i.

By solving for the optimal allocation, the goal is to minimize the value of $\lambda = \max\{S_{cpu}, S_{gpu}\}$, which represents the maximum load across the system. Although solving this load-balancing problem is NP-complete, the small number of operations involved makes it computationally feasible.

B. Weight placement

Weight placement is a critical component of optimizing LLM inference, especially when the model's parameters exceed the GPU's memory capacity. Proper weight placement strategies help minimize memory transfer overhead, reducing latency and improving throughput.

In FlexGen, weight placement is managed at the layer granularity. The method pins parts of the weights for each layer to the GPU memory, ensuring that frequently accessed weights are immediately available to the GPU during inference. This approach minimizes the need for frequent data transfer between CPU and GPU memory, which can introduce significant delays due to bandwidth limitations.

- Layer Granularity: Only certain parts of each layer's weights are pinned to GPU memory. This allows for a more flexible and efficient use of the available GPU memory.
- Zig-Zag Strategy: A technique called zig-zag is used to reuse weights within the same column across layers, further optimizing memory usage. This strategy reduces the memory footprint by ensuring that weights are reused efficiently without redundant transfers.

TwinPilots introduces a more dynamic approach to weight placement by using both host memory and GPU memory. The strategy relies on sparse memory access patterns to improve memory efficiency (as shown in Fig. 10).

- Embedding Layers: The embedding layers are assigned to host memory. These layers typically involve sparse memory access patterns, meaning that they don't require the full weight matrix to be loaded at once. Instead, a row-indexing technique is employed, which allows only the necessary rows to be accessed during each operation. This reduces the amount of memory needed on the GPU.
- Transformer Layers: For transformer layers, pinned weights are placed in GPU memory. The first few transformer layers, which are computation-heavy, are placed in the GPU for fast execution. However, as the model scales, the un-pinned weights are dynamically loaded into host memory, allowing the GPU to focus on active computation while the CPU manages the larger portions of the model.

These strategies ensure that the system can handle very large models by effectively distributing the weight matrices across memory spaces, thereby reducing memory contention and increasing the processing efficiency of both the CPU and GPU.

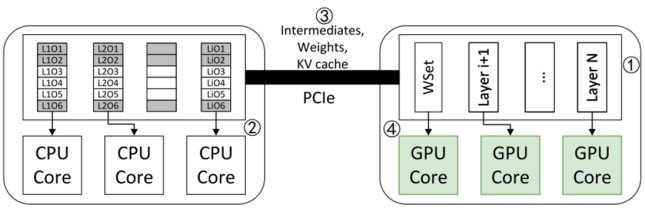


Fig. 10. Data placement of TwinPilots

C. KV Cache Placement

The KV cache is a crucial component in LLMs, especially in the decoding stage. It stores key-value pairs used during attention calculations, and its placement is critical for ensuring low-latency and high-throughput inference.

FlexGen employs a similar strategy for KV cache placement as it does for weight placement, managing cache at the tensor granularity. The strategy pins portions of the KV cache elements to the GPU, ensuring that the most frequently accessed parts of the cache are readily available to the GPU during decoding. This minimizes the need for constant memory transfers between CPU and GPU.

This approach allows for more fine-grained control (tensor granularity-based) over how the KV cache is placed, with different parts of the cache being pinned to GPU memory as needed during the inference process. Furthermore, FlexGen uses a policy search algorithm to determine the best placement for the KV cache, balancing the load between the CPU and GPU to avoid memory bottlenecks and latency issues.

In Llama.cpp, the KV cache is handled by pre-allocating memory to accommodate the maximum sequence length. This ensures that there is sufficient space to store the entire KV cache, but it may not always be the most efficient in terms of memory usage, especially for shorter sequences.

Llama.cpp reserves a fixed amount of memory for the KV cache based on the maximum sequence length, which simplifies memory management but may lead to wasted memory in scenarios where the sequence length is much shorter than the maximum capacity. However, in some cases, Llama.cpp dynamically adjusts the size of the KV cache, growing it as needed based on the current decoding requirements. This reduces wasted memory but adds some complexity to the cache management.

Dynamically allocating is an efficient and sophisticated method to manage data. vLLM [20] is a wonderful example. It takes an excellent approach to KV cache management by dynamically allocating and managing the KV cache. Fig. 11 shows an example. This dynamic allocation is done at the block granularity, similar to how memory management works in operating systems using virtual memory pages. The

TABLE II
HARDWARE SETTINGS.

CPU	AMD EPYC 7R32 2.8GHz, 24 cores [1]
SIMD	AVX256 [10]
Memory	256GB DDR4
GPU	4× NVIDIA A10G 24G GDDR6
Interconnect	PCIe 3.0×16

KV cache is divided into blocks, and each block is managed independently. This allows for more efficient memory allocation and deallocation, reducing the chance of memory fragmentation.

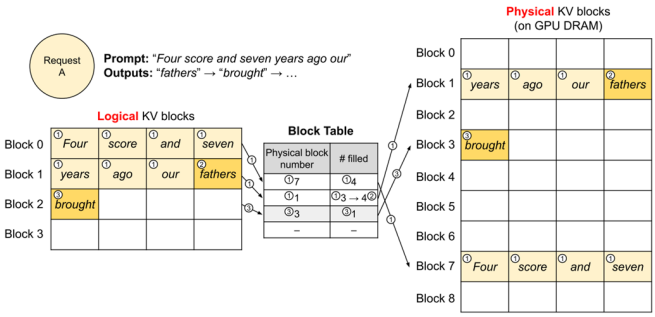


Fig. 11. Block table translation in vLLM

Similar to vLLM, TwinPilots introduces the idea of dynamically allocating and managing the KV cache with block granularity. The dynamic allocation of the KV cache enables better memory utilization during inference, as the system can adapt to varying memory needs depending on the current sequence length and workload. TwinPilots uses FIFO buffers to store partial KV cache blocks on the GPU. This helps to maintain a steady flow of data between the CPU and GPU, preventing the system from being bottlenecked by memory transfer delays. The system can allocate additional memory for the KV cache as required, based on the current sequence length and decoding stage. This dynamic allocation ensures that memory is used efficiently, particularly for long sequences.

V. EVALUATION

This section evaluates the above three methods' performance, and presents corresponding analysis for experiment results.

The evaluation is conducted on a system with the following configurations, and hardware setting is shown in table II:

- CPU: PyTorch 2.3 with Intel-MKL and OpenMP for parallelism.
- GPU: Linux 6.5 with CUDA 12.1.
- LLM Models: The models used for evaluation include OPT-30B, OPT-66B, and Llama-30B, all represented in FP16 format, with varying batch sizes and sequence lengths.

This setup ensures that the three methods can be fairly compared on the same hardware and software platform, simulating real-world deployment conditions for large-scale LLMs.

A. Decoding Throughput

The decoding throughput is a critical metric for evaluating inference performance. It is calculated as the ratio of the decoding length to the decoding time, which reflects how quickly the system can process a sequence.

FlexGen operates with a pipeline inference across three consecutive transformer layers. This design allows the model to process multiple layers in parallel, reducing the overall inference time. However, its throughput performance heavily depends on the efficiency of the pipeline and the memory bandwidth between CPU and GPU.

Llama.cpp exhibits high throughput for smaller batch sizes and shorter sequence lengths, as it minimizes the communication overhead between CPU and GPU. However, as the batch size and sequence length increase, Llama.cpp's throughput drops sharply. This is due to the increasing CPU bottleneck and insufficient GPU memory, which becomes strained as larger models and batches are used.

TwinPilots consistently outperforms FlexGen in terms of throughput. This improvement is attributed to higher CPU utilization, which reduces bottlenecks caused by GPU underutilization. Despite the unchanged GPU execution time, the additional CPU load results in higher throughput due to better load distribution and efficient resource utilization.

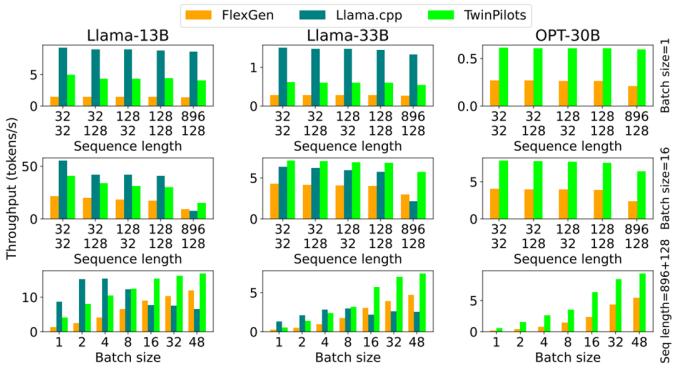


Fig. 12. Decoding throughput on various LLMs

Fig. 12 shows the results, where TwinPilots provides a significant improvement in throughput over FlexGen due to better load distribution between CPU and GPU. While FlexGen relies heavily on GPU-centric pipelining, TwinPilots leverages both CPU and GPU in parallel, allowing for better overall system utilization.

Llama.cpp excels in throughput for smaller batches and sequence lengths. Its optimized CPU-GPU communication allows it to outperform other methods in scenarios with lower computational demand. However, TwinPilots consistently delivers better results across a range of batch sizes and sequence lengths. While Llama.cpp's throughput decreases significantly with larger batches (due to CPU bottlenecks), TwinPilots maintains its performance due to dynamic scheduling and efficient load balancing.

Llama.cpp experiences significant increases in response latency when batch sizes exceed four (as shown in Fig.

13). For very large batch sizes, the latency can extend to several hours, as the system struggles to manage both CPU and GPU resources effectively. While TwinPilots consistently achieves much lower response latency across all batch sizes. Its ability to dynamically allocate resources between CPU and GPU results in more efficient inference processing and faster response times, even as batch sizes grow.

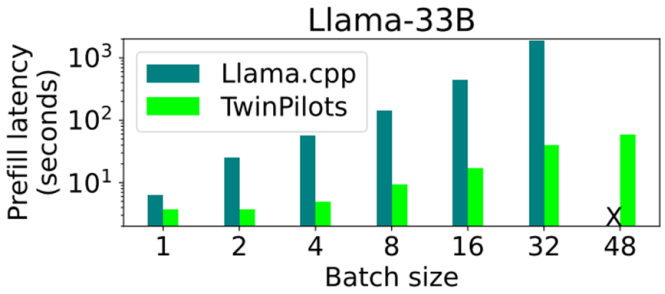


Fig. 13. First token latency comparison betweenLlama.cpp and TwinPilots

B. CPU-GPU Load Balance

The evaluation also measures the CPU-GPU load balance, which reflects how evenly tasks are distributed between the CPU and GPU. A well-balanced load distribution leads to more efficient utilization of the system's resources.

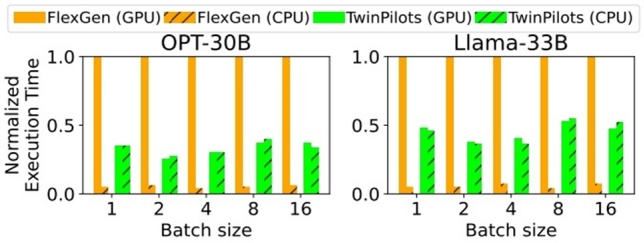


Fig. 14. GPU-CPU load (prompt length = 896, decode length = 128)

The result for loading balance has been demonstrated in Fig. 14. It is obvious that TwinPilots achieves a balanced load distribution by dynamically adjusting the allocation of tasks based on real-time statistics. The system adjusts the workload between the CPU and GPU so that neither component is underutilized or overwhelmed.

C. Multiple GPU Evaluation

The evaluation also includes tests on multiple GPUs, specifically Llama-70B with both pipeline model parallelism and tensor model parallelism configurations. The primary focus is on scalability and how well the methods can handle increasing data loads across multiple GPUs.

For FlexGen each GPU handles several consecutive transformer layers. This method has limited scalability, as each GPU is assigned a fixed portion of the model. While it works well with multiple GPUs, it faces challenges in handling very large models efficiently due to the fixed layer assignments.

TABLE III
COMPARISON OFUSER RESPONSE LATENCY FOR MULTIPLE GPUS (IN SECONDS). BATCH SIZE=4.

#GPUs	FlexGen	TwinPilots-PP	TwinPilots-TP
1	11.799	5.435	5.952
2	8.154	5.287	3.454
3	7.290	4.386	2.429
4	7.185	3.566	2.033

TwinPilots uses pipeline model parallelism (PP) and tensor model parallelism (TP). In PP, each GPU processes a portion of the model's layers, while in TP, tensors are partitioned along the row or column dimension, enabling finer granularity. This dynamic approach allows for better scalability and flexibility, reducing idle times and improving load balancing across GPUs.

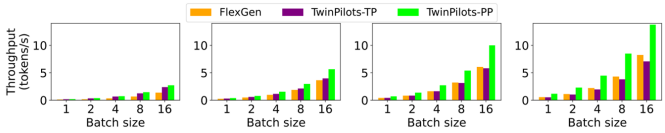


Fig. 15. Throughput for Multi-GPU on Llama-70B.

From Fig. 15 we can find, TwinPilots-PP offers higher throughput and better scalability compared to FlexGen. This is because it transfers less data between GPUs, thanks to the efficient partitioning and memory management.

What's more, TwinPilots-TP demonstrates lower response latency (Table III) due to finer-grained tensor partitioning, which reduces idle time and ensures faster processing of each batch.

VI. CONCLUSION

From the experimental results, we can emphasize the importance of balancing heterogeneous systems for efficient LLM inference. The key takeaway is that leveraging both GPU and CPU collaboratively enhances performance, particularly for large models. On the other hand, the GPU-CPU communication bandwidth remains a significant bottleneck. A new architecture with shared memory management and novel data transfer methods is proposed to address this issue. We have witnessed the potential of the TwinPilots approach, which optimizes resource utilization, minimizes latency, and ensures scalability, making it a promising solution for high-performance LLM inference.

REFERENCES

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In Proceedings of the 31st International Conference on Neural Information Processing Systems (Long Beach, California, USA) (NIPS'17). Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[2] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. OpenAI blog 1, 8 (2019), 9.

[3] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. Journal ofMachine Learning Research 24, 240 (2023), 1–113.

[4] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, MarieAnne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971 (2023).

[5] Machine Learning Compilation. 2024. MLC LLM. https://llm.mlc.ai/

[6] BentoML. 2024. OpenLLM. https://github.com/bentoml/OpenLLM

[7] Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Levskaya, A., Heek, J., Xiao, K., Agrawal, S., and Dean, J. Efficiently scaling transformer inference. arXiv preprint arXiv:2211.05102, 2022

[8] NVIDIA. Fastertransformer. https://github.com/NVIDIA/FasterTransformer, 2022.

[9] Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.G. Orca: A distributed serving system for TransformerBased generative models. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pp. 521–538, 2022.

[10] Wang, X., Xiong, Y., Wei, Y., Wang, M., and Li, L. Lightseq: A high performance inference library for transformers. In Proceedings ofthe 2021 Conference ofthe North American Chapter ofthe Association for Computational Linguistics: Human Language Technologies: Industry Papers, pp. 113–120, 2021.

[11] Fang, J., Yu, Y., Zhao, C., and Zhou, J. Turbotransformers: an efficient gpu serving system for transformer models. In Proceedings ofthe 26th ACM SIGPLAN Symposium on Principles and Practice ofParallel Programming, pp. 389–402, 2021.

[12] Aminabadi, R. Y., Rajbhandari, S., Awan, A. A., Li, C., Li, D., Zheng, E., Ruwase, O., Smith, S., Zhang, M., Rasley, J., et al. Deepspeed-inference: Enabling efficient inference of transformer models at unprecedented scale. In 2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 646–660. IEEE Computer Society, 2022.

[13] HuggingFace. Hugging face accelerate. https://huggingface.co/docs/accelerate/index, 2022.

[14] Borzunov, A., Baranchuk, D., Dettmers, T., Ryabinin, M., Belkada, Y., Chumachenko, A., Samygin, P., and Raffel, C. Petals: Collaborative inference and fine-tuning of large models. arXiv preprint arXiv:2209.01188, 2022.

[15] Hoefler, T., Alistarh, D., Ben-Nun, T., Dryden, N., and Peste, A. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. J. Mach. Learn. Res., 22(241):1–124, 2021.

[16] Kwon, S. J., Kim, J., Bae, J., Yoo, K. M., Kim, J.-H., Park, B., Kim, B., Ha, J.-W., Sung, N., and Lee, D. Alphatuning: Quantization-aware parameter-efficient adaptation of large-scale pre-trained language models. arXiv preprint arXiv:2210.03858, 2022.

[17] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-Throughput Generative Inference of Large LanguageModels with a Single GPU. In Proceedings ofthe 40th International Conference on Machine Learning (Honolulu, Hawaii, USA) (ICML'23). JMLR.org, Article 1288, 23 pages.

[18] Huang, C.-C., Jin, G., and Li, J. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 1341–1355, 2020.

[19] Jia-Wei, H. and Kung, H.-T. I/o complexity: The red-blue pebble game. In Proceedings of the thirteenth annual ACM symposium on Theory ofcomputing, pp. 326–333, 1981.

[20] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. arXiv:cs.LG/2309.06180

[21] Aminabadi, R. Y., Rajbhandari, S., Awan, A. A., Li, C., Li, D., Zheng, E., Ruwase, O., Smith, S., Zhang, M., Rasley, J., et al. Deepspeed-inference: Enabling efficient inference of transformer models at unprecedented scale. In 2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 646–660. IEEE Computer Society, 2022.

[22] Chengye Yu, Tianyu Wang, Zili Shao, Linjie Zhu, Xu Zhou, and Song Jiang. 2024. TwinPilots: A New Computing Paradigm for GPU-CPU Parallel LLM Inference. In Proceedings of the 17th ACM International Systems and Storage Conference (SYSTOR '24). Association for Computing Machinery, New York, NY, USA, 91–103.